Dynamically Estimating Reliability in a Volunteer-Based Compute and Data-Storage System

Muhammed Uluyol

University of Minnesota

Abstract

Although cloud computing is a powerful tool for analyzing large datasets, it is not appropriate tool for highly distributed data. In such cases, moving data to a central location incurs high network overhead. To avoid this, Nebula distributes computation over a wide-area network while trying to maintain locality to data. However, Nebula has few mechanisms for creating redundant copies for computational tasks. We introduce a scheduler that preemptively creates redundant copies of tasks reducing the average task execution time at the cost of overall throughput in the case where the system sees no failures.

1 Introduction

Cloud computing has established itself as the de facto way to provide many types of online services and conduct data analysis. However, data is often being produced in a distributed fashion, and moving it to a central location for analysis is expensive. For example, conducting log analysis on logs that generated around the globe incurs both high latency and bandwidth consumption. Systems which distribute computation to be close to the data are more appropriate for such workloads. Nebula is a system that distributes computation and storage tasks to volunteers that are connected over a wide-area network [9].

Nebula consists of a variety of centralized components that manage resources, schedule tasks, and queue jobs. Users post applications to Nebula which are broken up into jobs (e.g. map and reduce) which are then broken up into tasks that are then executed on compute nodes. Volunteers join the system as either a compute or data storage (datastore) node, but can optionally serve both tasks. The volunteer nodes are inherently an unreliable resource. Volunteer resources may be removed from the system at any moment, which could force the system to re-execute work. Because both computation and data storage are done using volunteer resources, this exasperates the issue of fault-tolerance further, and questions arise with regards to the interactions between data storage and computation. Nebula has been designed to execute many types of jobs, but the ones most often run on Nebula are MapReduce jobs [5]. For these, the primary concern with losing datastore nodes is that the data lost will have to be recomputed (if possible) and losing compute nodes requires tasks to be rescheduled. However, it is also desirable for the system to maintain good data locality between the compute and datastore nodes that are communicating with each other in order to maximize performance.

Nebula has limited support for redundancy in data storage. Nebula will maintain three replicas for all data stored in the system. This is akin to GFS which was designed to store data in a centralized cluster environment [6]. Such environments are very different from that of Nebula. Ideally, enough replicas should be maintained so that data is Byzantine fault-tolerant [8] which has been done is systems like Glacier [7]. However, maintaining few replicas reduces overhead and may be sufficient for short-lived data (e.g. intermediate results) so a hybrid mechanism would likely take better advantage of resources.

Originally, Nebula only redundantly scheduled compute tasks when spare compute nodes were available. This may seem to maximize performance, but stragglers often act as performance bottlenecks [5]. Additionally, because volunteers may leave the system without warning, failures are also common. So in order to increase the consistency in computational time, it is important to schedule tasks on multiple compute nodes. Once one node completes, we may cancel the redundant tasks as they are no longer necessary.

1.1 Contributions

Our contributions are as follows:

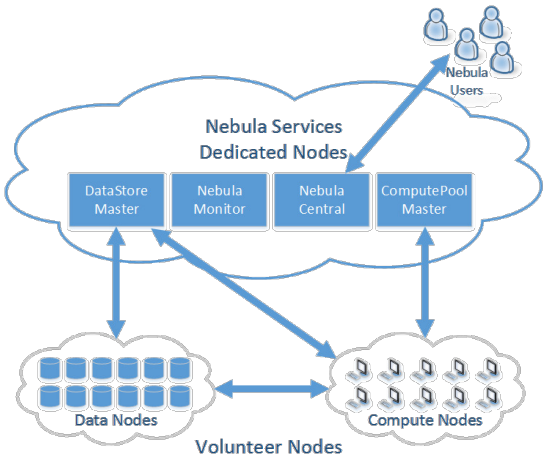


Figure 1: Nebula Architecture

1. We develop a notion of reliability for compute nodes and maintain estimates of this for all compute nodes.

2. We develop a mechanism to schedule tasks to run on a group of compute nodes so that they may meet a target reliability level.

We find that our solution incurs some overhead while decreasing average per-task computational time. We also find that our solution executes fewer tasks simultaneously while maintaining similar utilization rates.

2 Nebula System Architecture

As illustrated in Figure 1, all compute and datastore nodes in Nebula are volunteer resources that are inherently unreliable whereas Nebula Central, Monitor, DataStore Master, and ComputePool Master are dedicated, reliable services running in a centralized location. As mentioned earlier, since any user can donate their computational and storage resources, it is possible that a volunteer node will function both as a compute and data node.

- *Nebula Central* is the front-end layer which allows users to interact with the system. It provides a web-based portal to join the system as a compute or datastore node, upload files into the datastore, and post applications to run on Nebula.

- *Nebula Monitor* provides health-checking and network performance monitoring services for all compute and datastore nodes in Nebula. This information is updated periodically and used by the ComputePool Master and DataStore Master for node selection to run tasks and upload files respectively.

- The *DataStore Master* tracks the metadata of all files and remaining storage available within datastore nodes. It also receives periodical updates from Nebula Monitor about which datastore nodes are currently online, the locations of online datastore nodes, and the bandwidth of datastore nodes. This information is used by the DataStore Master to determine which data nodes should be assigned when a user upload files to Nebula. Datastore nodes provide put and get APIs to access the data.

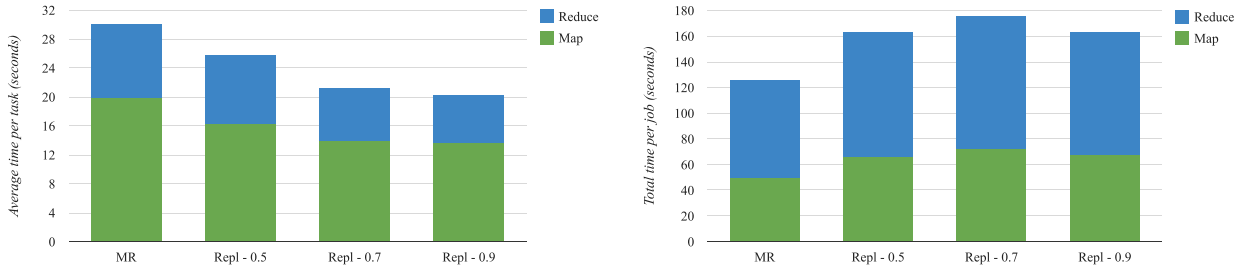
- The *ComputePool Master* performs task scheduling for the compute nodes. Like to the DataStore Master, it receives periodical updates from the Nebula Monitor with health and bandwidth information about compute nodes. When scheduling tasks, the ComputePool Master will attempt to maximize data locality and avoid selecting low-performance nodes. Compute nodes execute tasks within a Native Client (NaCl) sandbox [11] to isolate volunteers from any potential malicious users of Nebula. However, using NaCl provides other challenges as it drastically reduces the ability of Nebula code to request system information. This in turns makes it difficult to measure compute performance and monitor system activity (which is useful for identifying when the system is under light load).

Nebula MapReduce is distinct from from standard MapReduce in that map tasks do not send data directly to reduce tasks. Rather, they write the intermediate results to the datastore. In the original MapReduce, map tasks communicate directly with reduce tasks to avoid the overhead of writing to a distributed filesystem [5]. However, the map tasks were assumed to be running on hardware that is much more reliable than what Nebula uses, so this optimization would provide no benefit to Nebula.

3 Design

The ComputePool Master will only schedule tasks to execute redundantly when there are unused compute nodes after scheduling all of the queued tasks. This is not ideal because it does not gracefully handle stragglers or complete node failures gracefully. The authors of a previous paper developed a solution [4] for the same problem in BOINC—another system that uses volunteer compute resources, but is different in that it has centralized data storage [1]. We have adapted and extended the approach used in the paper for use in Nebula.

For every compute node, we maintain a reliability estimate. This number represents the probability of receiving a correct response from the node within a timely



(a) Average Task Completion Time

(b) Total Runtime

Figure 2: Task execution times on the original and modified schedulers

manner. This is computed by maintaining a counter for the number of tasks scheduled on that node and a counter for measuring the number of tasks that have failed or timed out. We then compute the number of successful tasks and divide this by the total to find the reliability of the node. As a result, this also measures performance to some extent. For a given task, we then select compute nodes using the Random-Fit algorithm found in [4] with a small extension: we measure the standard deviation of the reliability scores and place bounds on the values. If we let $b_{\max}$ represent the deviation we are willing to allow, then any nodes with reliability $< \mu - b_{\max}\sigma$ will not be scheduled on, and reliability scores will be capped at $\mu + b_{\max}\sigma$. We do this so that nodes will very low reliability scores are ignored and so that nodes cannot have overly optimistic reliability scores to avoid unrealistic values that are likely an artifact of our estimation mechanism.

4 Evaluation

We have deployed Nebula on PlanetLab [2], where we allocated 8 nodes as computational resources and 8 nodes for data storage. Nebula Central, Monitor, DataStore Master, and ComputePool Master were all run on a dedicated machine with an Intel® Xeon® CPU E5-2609 and 16 GB of memory. The application that we executed was the standard MapReduce WordCount with a 128 MB file. The file was split into 10 MB chunks that were randomly distributed across the datastore nodes, and we created one task per chunk. For our modified scheduler, we enforced a minimum of two compute nodes per reliability group and a maximum of six. Additionally, we set $b_{\max} = 2$ to enforce that reliability scores be no more than two standard deviations away from the mean. We then compared the original scheduler with ours using target reliabilities of 0.5, 0.7, and 0.9. All results are averaged over five runs.

Figures 2-4 refer to the original scheduler as 'MR' and our modified versions as 'Repl.' As expected, Figure 2a demonstrates that the average runtime of a task has decreased. This suggests that in the face of stragglers, our scheduler will be able to maintain it's performance characteristics better than the original Nebula scheduler. However, as Figure 2b indicates, the time taken to complete jobs has gone up substantially. This is a trade-off that can be made per-job by adjusting the parameters used in the scheduler.

In Figures 3 and 4, the reason for the dip near the middle of graph is that the map job was nearing completion and the reduce job could not begin until the map job completed. Figure 4 shows that the utilization of the schedulers is similar so they are using roughly the same resources. When taken in the context of Figure 3, it also tells us that the original scheduler uses its resources to run more tasks simultaneously whereas the new one uses them for redundancy.

Overall, these results indicate to us that the trade-offs made in the design show up as expected empirically. Thankfully, we also see that the average runtimes of tasks have decreased which was our goal.

5 Related Work

Other work includes BOINC, a system to use volunteer nodes for doing large-scale computation [1]. Unlike Nebula, BOINC does not take locality into account with regards to computation placement nor does it store data on volunteer systems. Additionally, BOINC does not run applications in a strong sandbox the way Nebula uses NaCl, and so it has more information about the underlying hardware and can use this in scheduling decisions. Nebula cannot probe the underlying system for much information and, as a result, information about the hardware must be found through other means.

Previous work in meeting reliability targets on volunteer nodes includes RIDGE [4]. RIDGE provided BOINC with a scheduler similar to that developed in this paper. However, due to the lack of a strong sandbox and the use of centralized data storage, the results of the

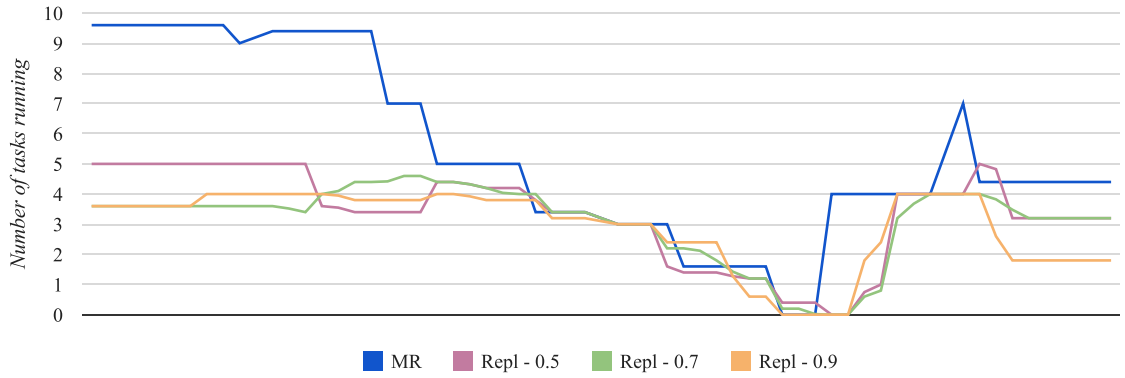


Figure 3: Tasks being executed simultaneously over the execution of the application

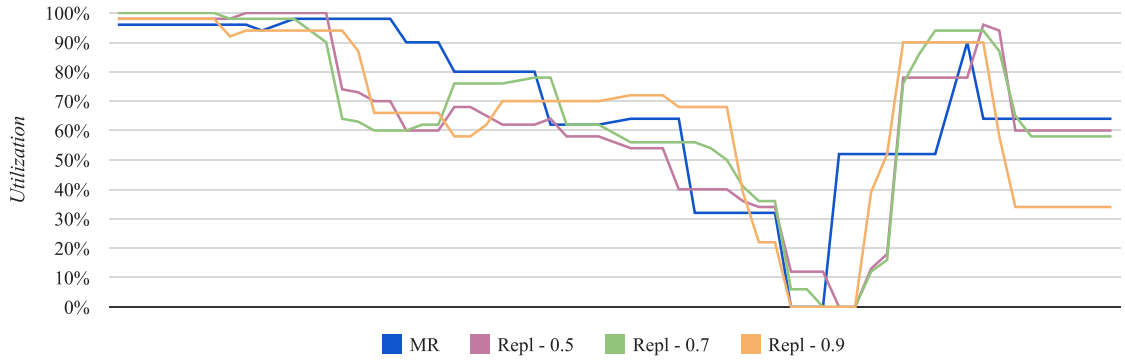


Figure 4: Compute nodes being utilized over the execution of the application

RIDGE paper cannot be directly applied to Nebula.

Other work in scheduling on wide-area distributed systems includes solutions for multi-cluster systems like G-Hadoop [10]. These systems assume the existence of traditional cloud clusters which are linked together over a wide-area network. Although related, this problem is distinct from the one being addressed by Nebula.

6 Future Work

In Section 1, we noted that Nebula has limited support for replication in data storage. Nebula creates three copies of data, but this may be too few in many cases or excessive in the case of intermediate data. Extending this to be Byzantine fault-tolerant for data that should persist may reduce data loss and the number of task re-executions. This can be done in a manner similar to the cloud-of-clouds storage backend in SCFS [3]. However, Byzantine fault-tolerance comes with high overhead even with the use of Reed-Solomon codes, so clients should have some mechanism of indicating whether this is desirable. An use-case for this is Nebula MapReduce where intermediate data is deleted soon after it is created.

7 Conclusion

Cluster computing has become the de facto platform for running data intensive compute jobs. While this system fits the needs of many applications well, there are applications which incur high overhead in this computing platform. In particular, analysis on data that is widely distributed is inefficient. Nebula is a cloud infrastructure that explores the use of volunteer resources to solve this issue. While Nebula uses data locality to reduce the runtime of jobs, it does little to prevent failures from affecting the performance of the system.

It is important to speculatively create redundant copies of computational tasks to reduce average runtime per task. On the other hand, redundant computation adds overhead to the system that reduces overall throughput. We introduced a scheduler that maintains estimates of reliability for all compute nodes and schedules tasks to meet reliability targets. Having the target reliability be configurable makes it simple to adjust based on the needs of particular jobs. By using the notion of reliability, we were able to avoid placing static requirements on the number of redundant copies to execute allowing us to balance redundancy and best-case throughput.

References

[1] ANDERSON, D. P. Boinc: a system for public-resource computing and storage. In *Proceedings of IEEE Grid Computing* (2004).

[2] BAVIER, A. C., AND ROSCOE, T. Operating systems support for planetary-scale network services.

[3] BESSANI, A., MENDES, R., OLIVEIRA, T., NEVES, N., CORREIA, M., PASIN, M., AND VERISSIMO, P. Scfs: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 169–180.

[4] BUDATI, K., SONNEK, J., CHANDRA, A., AND WEISSMAN, J. Ridge: combining reliability and performance in open grid platforms. In *Proceedings of the 16th international symposium on High performance distributed computing* (2007), ACM, pp. 55–64.

[5] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[6] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 29–43.

[7] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 143–158.

[8] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982), 382–401.

[9] RYDEN, M., OH, K., CHANDRA, A., AND WEISSMAN, J. Nebula: Distributed edge cloud for data intensive computing. In *IC2E 2014: IEEE International Conference on Cloud Engineering* (2014).

[10] WANGA, L., TAO, J., RANJAN, R., MARTEN, H., STREIT, A., CHEN, J., AND CHEN, D. G-hadoop: Mapreduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems 29* (March 2013), 739–750.

[11] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGA, N. Native client: a sandbox for portable, untrusted, x86 native code. In *Proceedings of IEEE Security and Privacy* (2009).